Aleksandar Nikolić

Eksploatacija buffer overflowa na x86_64 Linux platformi

Na x86_64 procesorskoj platformi je uvedena hardverska podrška za neizvršne memorijske stranice, takozvani No Execute bit (NX bit), i promene u načinu rada ELF izvršnih datoteka učinile su dosadašnje tehnike eksploatisanja buffer overflowa beskorisnim. Različite Linux distribucije imaju dodatne proaktivne zaštite kao što su ASLR (Adress Space Layout Randomization) ili ExecShield koje još više otežavaju proces eksploatacije. NX osigurava da ne postoje starnice memorije koje su istovremeno označene kao writeble i executable što klasičnu šelkod eksploataciju čini nemogućom. ASCII armoured adrese funkcija u bibliotekama onemogućavaju klasičnu ret-into-libc tehniku. Nasumično raspoređivanje steka i hipa osigurava drugačiji izgled memorije procesa pri svakom pokušaju eksploatacije. U radu je opisan način eksploatacije klasičanog stek overflow baga, ali se tehnika može proširiti i na eksploataciju heap overflowa. Razvijena tehnika omogućava pouzdanije eksploatisanje buffer overflowa s ciljem da se otkrivanjem slabosti postojećih sistema zaštite unapredi sigurnost istih. Radna platforma je Fedora Core 7 Linux operativni sistem na x86_64 arhitekturi sa omogućenim ASLR, ExecShield i NX zaštitama, ali se ovakav način zaobilaženja zaštite može primeniti i na ostalim sličnim sistemima.

Uvod

Buffer overflow bagovi predstavljaju veliku pretnju po sigurnost računarskih sistema. Na x86 arhitekturi procesora memorijske stranice označene sa *read* su takođe i izvršne. To je bila osnovna ideja kod klasične eksploatacije buffer overflowa. Dodatni kernel patchevi su doneli softversku zaštitu u vidu neizvršnih memorijskih stranica [2].

Jedna od novina kod x86_64 arhitekture jeste to što *write* oznaka memorijskih stranica sa sobom ne povlači i executable. Dakle postoje hardverski podržane non executable write stranice. Return into libc [3] tehnika je razvijena radi zaobilaženje ovog vida zaštite. Za razliku od klasičnog načina eksploatacije šelkodom, izvršavanje programa se ne nastavlja na šelkodu već se vraća u libc (standardna biblioteka koja sadrži osnovne funkcije). Izvršavanje programa se preusmerava na neku od funkcija, a argumenti funkciji se prosleđuju na stek. Na Linux x86_64 sistemima CPU se nalazi u takozvanom *long* modu. Registri opšte namene (engl. *General purpose registers*) nose 64-bitne vrednosti i memorijske adrese su 64-bitne. .data sekcija je mapirana RW, .text sekcija RX, a stek RW. Dakle, ne postoje stranice u koje može da se piše, a koje su ujedno izvršne.

Klasična return into libc tehnika ne funkcioniše na x86_64 arhitekturi jer prema ELF64 SystemV Application Binary Interface specifikacijama [4] argumenti funkcijama se prosleđuju preko general purpose registara. Da bi return into libc tehnika radila u *%rdi* registru se mora nalaziti prvi argument funkcije, dakle proširenje tehnike je potrebno. Odgovarajuća vrednost za prvi argument funkciji se mora smestiti u *%rdi* registar. Borrowed code chunks [1] tehnika je unapređenje return into libc tehnike koja pozajmljujući kod iz memorije procesa postavlja odgovarajuće vrednosti u odgovarajuće registre. Potom se poziva funkcija iz libca potpuno isto kao u return into libc tehnici. Na 2.6 grani Linux kernela uvdena je ASLR zaštita od eksploatacije buffer overflowa.

Nasumično raspoređivanje nekih delova memorije procesa otežava buffer overflow eksploataciju, ali je moguć brute force napad koji zaobilazi ovu zaštitu. Problem ASLRa na 32-bitnim sistemima je u

Aleksandar Nikolić (1988), Aleksinac, Savska 7, učenik 4. razreda Aleksinačke gimnazije

ograničenosti broja bitova na kojima memorijski prostor može nasumično da se rasporedi [5]. Na x86_64 arhitekturi za nasumično raspoređivanje je dostupno daleko više bitova pa brute force napad tog reda veličine nije praktičan [5].

Dinamičko utvrđivanje adresa u libc

Cilj projekta je implementacija uspešne eksploatacije buffer overflowa pomoću NULL trerminated stringa na sistemu sa NXom i ASLRom što znači da ne postoji RWX deo memorije ciljnog procesa. Da bi se uspešno eksploatisao buffer overflow bez nasumičnog pogađanja potrebnih adresa (adresa simbola u libc, adresa samog bafera), potrebno je naći način za dinamičko određivanje istih.

Memorijska adresa na kojoj je učitan libc se bira nasumično. Svaki simbol u libc nalazi se na drugom mestu pri svakom pokretanju programa. (ExecShield za veće biblioteke koristi prelink, što znači da se adresa na kojoj se libc učitava menja periodično, ne sa svakim pokretanjem programa (radi optimizacije kod većih biblioteka) ali je princip nasumičnog učitavanja generalno isti za sve optimizacije). Kako je nasumično učitan ceo libc to znači da su ofseti između bilo koja dva simbola u libc konstantni. Pronalaženjem tačne adrese nekog simbola u libc i poznavanjem njegovog ofseta do drugog simbola, moguće je utvrditi adresu drugog simbola.

Svaka libc funkcija koju program koristi se poziva preko svog PLT unosa. Prvi put kada se funkcija poziva, poziva se dinamički linker kako bi odredio adresu te funkcije u libc, nakon toga ta adresa se smešta u GOT tabelu (Global offset table) i svi

Funkcija system() na različitim adresama

```
[ea@localhost my]$ gdb -q ls
Using host libthread_db library "/lib64/libthread_db.so.1".
(gdb) break main
Breakpoint 1 at 0x402820: file ls.c, line 1033.
(gdb) r
Starting program: /home/ea/projects/chunkee64/my/ls
[Thread debugging using libthread_db enabled]
[New Thread 46912496279264 (LWP 13561)]
[Switching to Thread 46912496279264 (LWP 13561)]
Breakpoint 1, main (argc=1, argv=0x7fffb4df94d8) at ls.c:1033
1033 {
(gdb) p system
$1 = {<text variable, no debug info>} 0x3ade60dcf0 <system>
(gdb) q
The program is running. Exit anyway? (y or n) y
[ea@localhost my]

[ea@localhost my]$ gdb -q ls
Using host libthread_db library "/lib64/libthread_db.so.1".
(gdb) break main
Breakpoint 1 at 0x402820: file ls.c, line 1033.
(gdb) run
Starting program: /home/ea/projects/chunkee64/my/ls
[Thread debugging using libthread_db enabled]
[New Thread 46912496279264 (LWP 17741)]
[Switching to Thread 46912496279264 (LWP 17741)]
Breakpoint 1, main (argc=1, argv=0x7fff0aa24108) at ls.c:1033
1033 {
(gdb) p system
$1 = {<text variable, no debug info>} 0x3e4740dcf0 <system>
(gdb)
```

budući pozivi te funkcije koriste pokazivač na funkciju u GOT tabeli.

Pozivajući funkciju preko njenog PLT unosa (koji se nalazi u .text segmentu) pokazivač na tu funkciju biće smešten u GOT tabelu. Ni PLT ni GOT nisu nasumično učitani u memoriju, tako da su te adrese specifične za svaki ELF izvršni fajl. Potrebno je zatim saznati taj pokazivač na funkciju jer on sadrži adresu same funkcije u libc. Zatim se na taj pokazivač dodaje ofset do simbola u libc koji je potreban za eksploataciju.

Sam algoritam za eksploataciju sastoji se iz sledećeg:

1) Pozivanje funkcije iz PLT unosta ELF izvršnog fajla
2) Uzimanje pokazivača na funkciju koji je smešten u GOT
3) Dodavanje ofseta do system() funkcije u libc
4) Nalaženje adrese u koju je smešten "/bin/sh" string
5) Postavljanje pokazivača na "/bin/sh" string u odgovarajući registar
6) Pozivanje system() funkcije

Sve korake algoritma treba implementirati koristeći isključivo delove koda koji se nalaze u .text segmentu ELF izvršnog fajla pošto je .text jedini izvršni segment koji nije nasumično raspoređen pri pokretanju programa. Instrukcije koje su potrebne za implementaciju algoritma su pop, ret, mov, add, sub i call.

Implementacija algoritma

Kako program koji se koristi za demonstraciju tehnike poziva u nekom trenutku malloc() funkciju, pokazivač na nju će se nalaziti u njegovoj PLT sekciji.

Bafer se popunjava da bi se prelio i *%rip* registar se prepisuje adresom malloc() unosa u PLT. Dinamički linker nalazi adresu maloc() funkcije u libc i smešta je u GOT tabelu. Na vrhu steka se nalazi pokazivač na mesto na koje je upisana adresa malloc() funkcije. Izvršenje se nastavlja na pop ebx instrukciji.

U .text segmentu su instrukcije oblika pop *%registar* jako česte, a praćene su i instrukcijama u kojima se koristi vrednost koja se nakon pop instrukcije nalazi u odgovarajućem registru, što pruža različite mogućnosti za sklapanje code chunkova koji su potrebni za uspešnu eksploataciju.

Nakon pop instrukcije u *%rbx* registru se nalazi pokazivač, tj. adresa na kojoj je prethodno upisana adresa malloc() funkcije u libc. Da bi bilo moguće izračunati adresu system() funkcije u libc, sama adresa malloc() funkcije se mora nalaziti u nekom registru. U programu korišćenom za demonstraciju tehnike nije nađena odgovarajuća instrukcija koja koristi *%rbx* registar, a kojom bi bilo moguće dodati ofset i izračunati adresu system() funkcije. Alternativa je mov (%rbx), rax instrukcija nakon čijeg se izvršenja u eax registru nalazi adresa malloc() u libc.

Pri pokušaju izračunavanja system() adrese javlja se problem. Kada bi se ofset do system() funkcije dodao na adresu malloc() funkcije u stringu koji eksploatiše program pojavio bi se NULL bajt. Stringovi u Cu su NULL terminirani, tako da bi taj NULL bajt prerano terminirao string, a to se mora izbeći da bi došlo do uspešne eksploatacije. Do toga dolazi jer je ofset do system() funkcije mali broj, oblika 0x123456, tako da prvi bajt mora ostati NULL. Kako je ofset među adresama funkcija u libc konstantan, adresu system() funkcije moguće je izračunati tako što se izračuna koliko treba da oduzeti od adrese malloc() funkcije (koja se nalazi na nižoj adresi od system() funkcije), zatim se od dinamički dobijene adrese malloc funkcije oduzme taj obrnuti ofset. Kako je ofset u suprotnom pravcu veliki, neće biti NULL bajta u njemu. Oduzimajući od male vrednosti jako veliku, napravi se krug oko unsigned brojeva i dobije se adresa system() funkcije. Sam obrnuti ofset se dobija tako što se u spoljašnjem, pomoćnom programu prvo izračunaju adrese system() i malloc() funkcija. Zatim se od vrha memorijskog prostora (tj. najvećeg 64-bitnog broja) oduzme adresa system() funkcije, dobijeni razultat se sabere sa adresom malloc() funkije i doda mu se 1. Oduzimajući tu vrednost od adrese malloc() funkcije do-

Unos za malloc() funkciju u PLT sekciji ELF izvršnog fajla

```
[ea@localhost my]$ objdump -j .plt -d ls | grep malloc
0000000000401870 <malloc@plt>:
[ea@localhost my]$
```

bija se adresa system() funkcije. Kako su ofseti konstantni, nasumično mapiranje libc u memoriju pri pokretanju spoljašnjeg programa nema nikakav uticaj na obrnuti ofset.

Za oduzimanje se može koristiti sub instrukcija i to u obliku sub *%rsi rax* zato što se vrednost rsi registra može kontrolisati sa steka, tj. može se proizvoljno postaviti. Za vrednost rsi se postavlja već izračunati obrnuti ofset koji je konstantan. Ovime se u *%rax* registru dobija adresa system() funkcije. Time je dinamičko utvrđivanje adrese završeno.

Jedina stvar koja je sada potrebna jeste adresa argumenta system() funkcije ("/bin/sh") bude postavljena u *%rdi* registar. U Aplication Binary Interface specifikaciji za 64-bitne ELF izvršne fajlove precizirano je da se argumenti funkcija nalaze isključivo u registrima opšte namene i to počevši od *%rdi* registra. Upravo zbog toga, za razliku od 32-bitnih ELF izvršnih fajlova, nije moguće postaviti argumente funkciji na stek.

Kada se adresa system() funkcije dobije u rax registru, trebalo bi je sačuvati kako se ne bi izgubila pri daljem podešavanju odgovarajućih adresa u registre upravo zbog ulaska u veći broj funkcija. Mesto za čuvanje utvrđenih adresa može biti .data segment memorijskog prostora programa. U kodu su jako česte instrukcije oblika mov *%rax, pointer*, gde pointer pokazuje na neki deo .data segmenta. Obično instrukcije ovakvog oblika imaju i par instrukcija koje čine obratno, mov pointer, %rax. Po potrebi se sačuvana može vrednost vratiti iz .data segmenta u odgovarajući registar.

Preostaje da se argument "/bin/sh" system() funkcije postavi negde u memoriji, a zatim da se pokazivač na taj deo memorije smesti u rdi registar. Kao što je sačuvana adresa system() funkcije, može se smestiti i "/bin/sh;" string u neki deo .data segmenta. Svaki string mora da bude terminiran NULL bajtom, dodatnim znakom ";" je terminirana shell komanda, a za uspešno eksploatisanje programa nije bitno šta se nalazi iza ";" znaka, odnosno gde je string terminiran. Druga mogućnost je da se koristi neki deo .data segmenta koji već jeste popunjen NULL bajtovima.

Kada je sve završeno trebalo bi nastaviti izvršenje na nekoj instrukciji oblika mov pointer, rdi gde pokazivač sadrži adresu argumenta system() funkcije. U programu koji je korišćen kao primer nije nađena odgovarajuća instrukcija. Alternativa je da se izvrši instrukcija oblika mov pointer, rax, a zatim druga instrukcija mov rax, rdi. Preostalo je još pozivanje instrukcije callq sa adresom system() funkcije kao argumentom. Kako, ponovo, nema direktno takve instrukcije u programu koji je korišćen kao primer mora se koristiti slična alternativa kao kod postavljanja adrese argumenta u odgovarajuć registar. Adresa system() funkcije se postavlja u rax registar i izršenje se nastavlja na callq *%eax instrukciji.

Time je proces eksploatacije uspešno završen bez nasumičnog pogađanja potrebnih adresa.

Prikaz uspešne eksploatacije buffer overflowa:

```
[ea@localhost my/puzzle]$> sh run
sh-3.00$ exit
exit
run: line 3: 19504 Segmentation fault
       ./ls-vuln "`./mkegg $(./symf)`"
[ea@localhost my/puzzle]$>
```

Napomene

Pri sklapanju različitih delova koda trebale bi se koristiti instrukcije što bliže prologu funkcije. Što je instrukcija koja se koristi bliža retq instrukciji koja je deo prologa funkcije, veće su šanse da se posledica instrukcije koja se koristi očuva do povratka iz funkcije. Retq instrukcije u stvari vezuju različite "pozajmljene" delove koda, a time je omogućena potpuna kontrola toka programa. Postavlja se pitanje da li će se u .text segmentu programa koji se eksploatiše naći sve potrebne instrukcije za uspešnu eksploataciju. Sama ta činjenica čini ovu tehniku eksploatacije specifičnu za svaki ELF fajl. Pri eksploataciji primera korišćen je pomoćni program koji prikazuje poslednjih 5 instrukcija svake funkcije u ELF fajlu koji se eksploatiše. Po tim instrukcijama moguće je naći više alternativnih tokova programa kojima bi bilo moguće implementirati dinamičko utvrđivanje adresa u libc. Sve veće Linux distribucije koriste sopstvene repositorye softvera, tako da se potpuno isti ELF faj-lovi nekog programa nalaze na svim sistemima koji koriste istu distribuciju što omogućuje razvijanje eksplojt programa koji je specifičan za distribuciju, odnosno za verziju programa koja se nalazi na repositoryu te distribucije.

U eksploitu programa koji je korišćen kao primer pozivana je system() funkcija sa samo jednim argumentom, ali je naravno moguće povezati veći broj funkcija sa većim brojem argumenata

uopštavanjem opisanog algoritma za dinamičko utvrđivanje adresa funkcija u libc. Jedna od mogućnosti je pozivanje mprotect() funkcije nad nekim delom memorije i menjanje dozvola na tom delu memorije u RWX, a zatim upisivanje klasičnog šelkoda na taj deo memorije i kasnije preusmerenje izvršenja na sam šelkod.

Potrebno je dosta vremena za vezivanje i praćenje raznih delova koda da bi se program uspešno eksploatisao. U [1] koristi se program koji automatski nalazi odgovarajuće delove koda. To je u ovoj situaciji moguće, ali vrlo ograničeno. Program u [1], koji automatizuje proces nalaženja delova koda koji se skupljaju, ima na raspolaganju čitav libc za odgovarajuće instrukcije, dok je u slučaju kada je libc nasumično mapiran u memoriji jedina mogućnost pretraga .text segmenta za odgovarajuće instrukcije. Kako se to razlikuje od slučaja do slučaja, morao bi se napraviti "pametniji" program za traženje alternativhin tokova za postizanje istog rezultata kako bi, u slučaju da nema odgovarajuće instrukcije, mogao da nađe drugi pravac izvršenja.

Mogući načini sprečavanja

Ova tehnika se oslanja isključivo na konstantnim adresama .text segmenta i konstantnim ofsetima između funkcija u libc. Moguće je prevodiocu zadati da kompajlira PIE (Position Independent Executable) ELF fajl koji kernel zatim pri svakom učitavanju može nasumično da smešta u memoriju (potpuno isto kao i deljenu biblioteku). Ovime se cela tehnika samo usporava jer je potrebno nasumično pogađati samo baznu adresu .text segmenta.

Zaključak

Korišćenjem ove tehnike mogu se uspešno zaobići ASLR, NX bit i ExecSheild zaštite od buffer overflowa na x86_64 sistemima. Tehnika omogućava razvijanje pouzdanih eksploita kojima nije potrebno nasumično pogađanje adresa.

Literatura

[1] Krahmer S. x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique. www.suse.de/čkrahmer/no-nx.pdf

[2] PaX. http:///pax.grsecurity.net

[3] Linux lpr buffer overflow exploit for non-executable stack. http://www.ouah.org/solarretlibc.html

[4] AMD64 Application Binary Interface (v 0.98). http://www.x86-64.org/documentation/abi-0.98.pdf

[5] Shacham H. On the Effectiveness of Address Space Randomization. Stanford University. http://www.stanford.edu/čblp/papers/asrandom.pdf

---

Aleksandar Nikolić

Bypassing Proactive Buffer Overflow Protections on Modern Architectures

Few new protection mechanisms were introduced on the x86_64 processor architecture, such as NoExecute (NX) or Advanced Virus Protection (AVR). New protections and different internals of ELF executable files have made old buffer overflow exploitation techniques useless. In addition, different Linux distributions have various additional protections like ASLR (Address Space Layout Randomization) and ExecShield which make exploitation even harder. There are no memory pages that are at the same time writable and executable if NX is used, which makes classical exploitation with shellcode impossible. The return into lib exploitation technique is also difficult to use because of the ASCII armored addresses of shared library functions. Random stack and heap memory mapping change the memory layout every time an executable is run. Our working platform is the Fedora 7 Linux operating system on an x86_64 architecture with ASLR, NX ExecSheild, and ASCII armored addresses protections enabled. The presented technique could, with some changes, be applied on other platforms. A stack overflow buf is used for technique demonstration, but it could be applied to heap overflows as well. The presented technique enables more reliable exploitation.